J Grid Computing (2009) 7:501–518
DOI 10.1007/s10723-009-9131-6

Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments

Eric Martin Heien · David P. Anderson · Kenichi Hagihara

Received: 20 February 2009 / Accepted: 12 August 2009 / Published online: 25 August 2009

Abstract Internet based volunteer computing projects such as SETI@home are currently restricted to performing coarse grained, embarrassingly parallel master-worker style tasks. This is partly due to the "pull" nature of task distribution in volunteer computing environments, where workers request tasks from the master rather than the master assigning tasks to arbitrary workers. In this paper we propose algorithms for computing batches of medium grained tasks with deadlines in pull-style volunteer computing environments. We develop models of unreliable workers based on analysis of trace data from an actual volunteer computing project. These models are used to develop algorithms for task distribution in volunteer computing systems with a high probability of meeting batch deadlines. We develop algorithms for perfectly reliable workers, computation-reliable workers and unreliable workers. Finally, we demonstrate the effectiveness of the algorithms through simulations using traces from actual volunteer computing environments.

Keywords Volunteer computing · Pull-style task distribution · Stochastic scheduling · Grid computing · Resource scheduling

E. M. Heien (✉) · K. Hagihara
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565-0871, Japan
e-mail: e-heien@ist.osaka-u.ac.jp

K. Hagihara
e-mail: hagihara@ist.osaka-u.ac.jp

D. P. Anderson
University of California, Berkeley, Berkeley, CA, USA
e-mail: davea@ssl.berkeley.edu

1 Introduction

In recent years, public-resource computing or "volunteer computing" projects have demonstrated the power of performing distributed computation using donated resources over the Internet. Projects such as SETI@home [1] and Folding@home [2] sustain computation speeds of tens or hundreds of teraflops, comparable with high end supercomputers. In these projects, independent computational tasks are distributed and executed on donated computer resources around the world.

Volunteer computing (VC) systems use a master-worker style of computing, where tasks are distributed from a master machine to worker machines and executed. Because these systems are composed of donated resources, they can make few guarantees about network or machine reliability. Therefore VC is usually applied to coarse grained embarrassingly parallel computation with

tasks that require hours or days to complete. Task completion deadlines are generally on the order of days or months because of the volatile nature of the donated resources.

VC environments differ from traditional grid computing environments in several important ways. First, because the worker machines in VC systems are owned by private individuals, communication and computation reliability is significantly lower than in most Grid systems. A worker machine may often be disconnected from the network, used for other purposes or completely quit the computation without advanced warning. Second, worker machines are often behind firewalls and use NAT (network address translation) techniques which only allow one-way worker to master connections. This means that a "pull" model of task distribution must be used, instead of the common "push" model where the master distributes tasks to arbitrary workers. Finally, worker machines provide virtually no resource reservation or querying capabilities, thereby making task scheduling difficult.

In this paper, we propose algorithms for computing batches of tasks with deadlines in VC systems given varying types of worker reliability. Rather than normal VC deadlines of days or months, we deal with deadlines of minutes or hours. We call this "low latency computing." Low latency computing in a VC system is appropriate for executing submitted batches of tasks with quick turnaround, or performing large scale barrier synchronous computations such as in the bulk synchronous parallel model [3]. Examples of such applications include molecular dynamics simulations with multiple trajectories, evolutionary based optimization algorithms with periodic swapping of solutions and any other problem with medium grained tasks and periodic barrier synchronizations. Applications such as Folding@home should have higher computing efficiency if they use a low latency style scheme. For pull-style task distribution in a VC system, the key to meeting batch deadlines is ensuring that all tasks are distributed to workers in a timely manner and that workers complete the tasks before the deadline. Because of the nature of VC systems, we use techniques similar to those from stochastic scheduling [4] to handle worker unreliability.

Previous studies investigated task distribution in grid and VC environments [5–7]. However, some of these assume work may be distributed to arbitrary workers, which is not valid for pull style environments. Others describe methods to maximize total system throughput rather than meet specific task deadlines. To the best of our knowledge, this paper is the first to investigate methods for computing low latency batches in a pull-style VC environment.

To develop suitable algorithms for low latency VC, we first define the environment and worker characteristics in Section 2. Using trace data from an actual VC environment, we show how worker task requests can be modeled as a Poisson process and task computation time can be predicted from past worker behavior in Section 3. From these models, we develop algorithms for task distribution in Section 4. The algorithms are verified using trace-driven simulations in Section 5. Finally, we review related work and offer our conclusions in Sections 6 and 7.

2 Computation Model

In this model for VC low latency batch computing, there are M batches of work, denoted $B_1, \ldots, B_M$. Each batch B_i has N independent tasks of equal computational size denoted $T_1^i, \ldots, T_N^i$, a submission time S_i and a deadline D_i with $S_i < D_i$. All tasks in batch B_i are available for distribution at time S_i. Batches are sequential and do not overlap, i.e. $\forall i, n, (i < n) \Rightarrow D_i < S_n$. Figure 1a graphically shows the model of tasks and batches used in this paper.

All tasks are initially on a single master server, which tries to assign tasks to workers so as to minimize the number of tasks completed after their deadline. A task that is completed before the batch deadline is called a satisfied task, and a batch whose tasks are all satisfied is called a satisfied batch. Note that batch submission and deadline times need not be predetermined, meaning that creation of the tasks in B_i can depend on completion of the tasks from B_{i-1}. In fact, due to

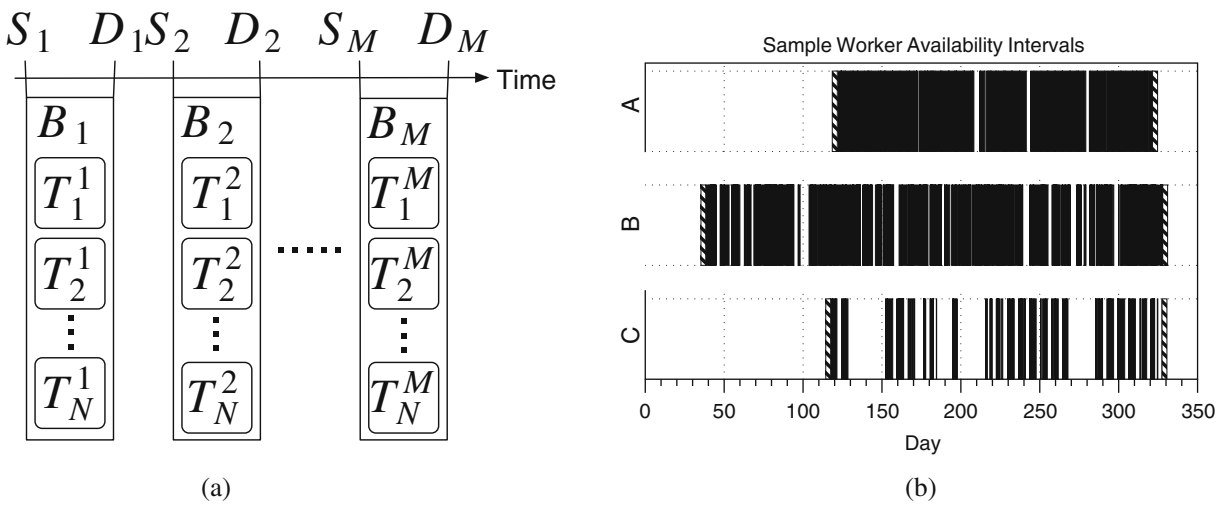

Fig. 1 Graphic depiction of tasks and batches in the low latency computing model, and three example workers with their availability intervals. **a** Batch diagram. **b** Availability intervals and worker lifespan

the unpredictable nature of VC systems, it is often difficult to predict S_i and D_i for batches far in the future.

To compute the tasks of a batch, there are P workers $W_1, \ldots, W_P$. In standard VC environments, connections may only be made from a worker to the master, not vice-versa. A connection is made when the worker initially becomes active, and at specified times afterwards called "reconnection times". At any given time, a worker is in one of two states — available or unavailable. The master is always available. Previous studies [8, 9] have shown this to be a good approximation of actual systems, rather than measuring availability as a fraction of available CPU power. A period where a worker is in an uninterrupted available state is called an availability interval, and a period where a worker is in an uninterrupted unavailable state is called an unavailability interval. The lifespan of a worker is defined as the time between the start of the earliest availability interval and the end of the latest availability interval. Average availability is the fraction of the worker lifespan spent in the available state. Figure 1b shows availability intervals for three workers with different average availability (A: 95%, B: 65% and C: 33%). Black sections indicate the worker is available and white sections indicate the worker is unavailable. The striped bars show the beginning and end of the worker lifespan. As seen in this figure, worker availability may be erratic and difficult to predict.

A worker in the available state may perform computation or communication, an unavailable worker may do neither. Various factors may cause transition between these states—user activity/idleness, machine reboot/shutdown, machine/network failure, etc. If a worker transitions from available to unavailable while executing a task, the task is resumed at the same point when the worker returns to the available state. This behavior can be achieved through task checkpointing. Each worker W_i also has a task computation time C_i, which is the number of seconds the worker requires in the available state to complete a task. This can be thought of as the inverse of the computation speed.

In this paper we treat C_i as a deterministic variable, though in real systems it might be more accurately treated as a probability distribution depending on the worker and task characteristics. For simplicity, we assume C_i is constant and that non-deterministic effects, such as swapping due to insufficient memory, can be avoided by using standard worker selection techniques. In addition, malicious or malfunctioning workers occurring in VC systems are not explicitly considered in this paper. These can be handled by computing redundant

tasks or other techniques such as ringers and magic numbers [10] or worker reputation measurement [11], though the usage of these is outside the scope of this paper.

In terms of communication and computation, workers can be termed either reliable or semi-reliable. A worker is "communication-reliable" if it can guarantee communication with the master at an arbitrary time R. A worker is "semi-communication-reliable" if it cannot guarantee communication at time R, but behaves like a standard VC worker. A worker W_i is "computation-reliable" if it can guarantee task completion within the task computation time of the worker (C_i). A worker is "semi-computation-reliable" if it cannot guarantee task completion within C_i, but behaves like a standard VC worker. Communication-reliable and computation-reliable workers are only theoretical constructs, but we use them as a basis to develop algorithms in later sections.

3 Analysis of Volunteer Computing Workers

In this section we examine workers from an actual VC environment and create models of them based on analysis. In Section 3.1 we describe the experimental data taken from a VC system used to model the workers. Analysis of worker availability is performed in Section 3.2. In Section 3.3 we examine communication reliability in workers and demonstrate that connections from VC workers can be modeled using a Poisson process. In Section 3.4 we examine computation reliability in workers, and derive a model for predicting worker computation reliability based on prior worker availability.

3.1 Worker Trace Data

To perform the analysis and experiments in this paper, we used a set of worker availability trace data. This trace data was measured using the Berkeley Open Infrastructure for Network Computing (BOINC) [12]. The BOINC middleware is used to perform large scale VC over the Internet. The BOINC client software currently runs on over 1 million computers throughout the world. The BOINC client was augmented to record the start and stop times of CPU availability on each worker machine. In BOINC, CPU availability is determined by user preferences and whether the machine is running. For example, some users only allow BOINC to run when the computer is idle or only on weekends, while other users shut down the machine every night.

Volunteer participants downloaded this BOINC client, which recorded the CPU availability intervals and reported the intervals back to the main server. A total of 112,268 worker machines ran the client during the period between April 1, 2007 to February 12, 2008, though none of the worker lifespans covered the entire period. In total, the modified client recorded 16,293 years worth of CPU availability. Among the worker operating systems, 66% ran Windows XP, 12% ran Windows Vista, 9% ran Mac OS X, 7% ran a variant of UNIX/Linux, and the remaining 6% ran a variant of Windows. Further analysis of the trace data is available in [13].

3.2 Analyzing Worker Availability

To learn more about the typical worker machine in a VC system, we performed several analyses of the trace data in regards to worker availability. These analyses were performed to confirm the validity of the trace data compared with previous studies, and to examine the characteristics of worker availability.

Figure 2a shows the number of available workers over the trace recording period. This was obtained by measuring the number of workers in the available state every 8 h over the entire trace period. The oscillation of available workers is caused by computers being shut down at night and on the weekends. Although this oscillation grows over time, the oscillation relative to the total number of workers stays relatively constant and doesn't affect the algorithms described here.

Figure 2b shows cumulative distribution plots of worker lifespan and worker availability. The point (x,y) on the dotted line means that x fraction of workers have lifespans less than y, and on the solid line means that x fraction of workers spend less than y of their lifespan in the available state. From this graph we see that only a small fraction

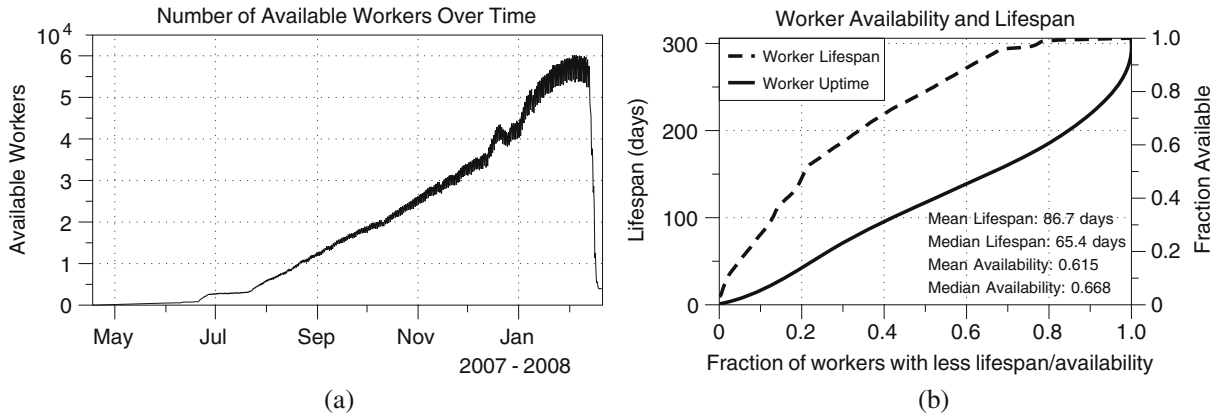

Fig. 2 Worker availability and lifespan characteristics. **a** Number of available workers. **b** Worker availability over trace period

of workers (roughly 5%) are available over 80% of their lifespan and that half of the workers are available for less than 40% of their lifespan. This environment of high unreliability and downtime makes traditional scheduling algorithms difficult to use.

In this paper we assume that worker unavailability is transient rather than permanent, in other words workers do not become permanently unavailable. For actual systems this is not a correct assumption due to machine failures, volunteers ending their participation, etc. An analysis of VC projects indicates worker lifespan roughly follows an exponential distribution with a mean of 3 months [14]. As seen in Fig. 2b, the mean and median worker lifespan are on the order of 2 to 3 months. This means that for low latency tasks less than 15 h in length, fewer than 1% of tasks will fail due to permanent worker unavailability. Therefore, we can simplify our model by ignoring permanent worker unavailability without significant effect.

To create algorithms for performing low latency computation in a VC environment, it is crucial to understand and model the behavior of workers in terms of availability and unavailability. Many studies have investigated worker availability in desktop grids [8, 9, 15, 16] and in VC type environments [13, 17, 18]. In the following two sections, we develop models of VC workers partly based on our own analysis and partly based on the work in these studies.

3.3 Modeling VC Worker Communication

In this section we examine the effect of worker unavailability on task requests and propose a model for task requests from VC workers. To develop a model of task requests from VC workers, we perform simulations using worker trace data. The simulation results indicate that task requests from VC workers can be modeled as a Poisson process. Furthermore, the task request rate of this process can be controlled through the worker reconnection period T. This means that rather than scheduling communication for individual workers, we treat the entire worker pool as a tunable stream.

To perform the simulations, we used a subset of 11,320 randomly selected workers from the entire trace data set. We performed a wide range of simulations to test multiple reconnection periods, worker pools and stretches of time. In a simulation, the workers transition between available and unavailable based on the trace data. Each worker W_j connects to the master at the start of their lifespan. After each connection, the worker is assigned a reconnection time $R_j = CurTime() + T$. If a worker is unavailable at time R_j, the connection is initiated when the worker next becomes available. Because the number of active workers changes over the course of the trace data, we define the number of recently active workers (P_{active}) as the number of workers which have connected in the last $2T$ seconds.

To analyze the results of the simulations, we recorded each time a worker connected to the master. These times were recorded during intervals $[S_0 + XL, S_0 + (X+1)L]$ for $X \in [0, M-2]$. This effectively emulates the task request behavior of the workers performing M batches. The parameters for the simulations are shown in Table 1.

The results of one simulation are shown in Fig. 3a with the time gaps between consecutive connections to the master plotted as a histogram and cumulative fraction. The parameters of this simulation are $T = 4$ h, $L = 1$ h, $S = 08$ Dec 2007, and at the start of the simulation $P_{active} = 4024$. Note that this represents connections from all workers, not from a single worker. For a reconnection time T with P_{active} active workers, the average expected rate of connection is one connection every T/P_{active} seconds. In Fig. 3a, the average time gap between connections of 3.76 s closely matches the expected time gap of 4 hours/4024 = 3.58 s. The cumulative distribution of time gaps in this simulation and others appears similar in shape to that of an exponential distribution function.

We made QQ and PP plots for the simulation shown in Fig. 3a. Figure 4 also shows QQ/PP plots for the same simulation. As seen, there is a close fit between the simulation results and an exponential distribution. There is a slight deviation for larger values as seen in the QQ plot (left). The PP plot (right) also shows a good fit. Based on these results we offer Hypothesis 1.

Hypothesis 1 *Let T be a positive time period. P workers with availability characteristics common to VC system workers are assigned reconnection times (T) as described above. Then the time gap between connections to the master can be modeled as an exponential distribution function (EDF).*

To confirm Hypothesis 1, we apply the Kolmogorov-Smirnov (KS) test [19] to all simulation results. This allows comparison of the simulation results with EDFs using the parameter based on the expected time between connections $\lambda = P/T$. The KS test is sensitive to imperfections in large data sets, so each KS p-value is based on the average from 100 random samplings of 100 data points each from each data set. Generally, the minimum acceptable p-value for the KS test is 0.05, so p-values higher than this indicate good fit with the EDF and the validity of Hypothesis 1.

Figure 3b shows the results of applying the KS test to the simulation results. In this figure, the x and y axes represent the reconnection time T and interval length L. The z axis represents the 5th percentile p-value over all simulations for a given T and L (i.e. 95% of simulations had a higher p-value than indicated in the graph). In other words, the graph shows the validity of Hypothesis 1 for a range of T and L. For example, with $T = 4$ h and $L = 2$ h, 95% of simulation results had a p-value greater than 0.22.

In this figure, the accuracy of modeling worker connection time gaps as an EDF varies depending on the ratio of T to L. If T is much greater than L, then few workers will connect in a given interval and the KS test will show a poor fit. For example, with $T = 4$ h, $L = 5$ min and $P = 1000$, an average interval will have only 4.2 connections even with full worker availability. For a lower ratio of T to L, worker connections fit an EDF very well.

A Poisson process [20] is defined as a process with time between events following an EDF. Since Hypothesis 1 showed that time gaps be-

Table 1 Task request simulation parameters

Parameter	Values
Number of workers (P_{total})	11320
Reconnection time (T)	5 m, 15 m, 1 h, 2 h, 4 h, 12 h, 1 day, 2 days, 4 days, 10 days
Interval lengths (L)	1 m, 5 m, 15 m, 1 h, 2 h, 4 h
Number of intervals (M)	200
Test period start times (S_0)	Sun, 01 Jul 2007, Wed, 01 Aug 2007, Mon, 01 Oct 2007, Sat, 01 Sep 2007, Thu, 01 Nov 2007, Sat, 01 Dec 2007, Tue, 01 Jan 2008

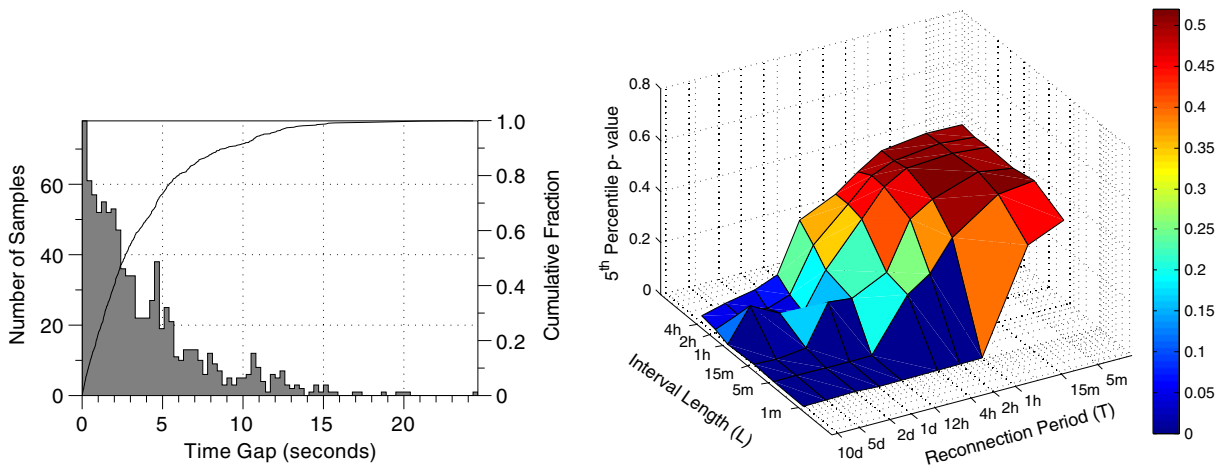

Fig. 3 Simulation results. **a** Time gaps between connections to the master. **b** p-vals from Kolmogorov-Smirnov test of EDF fit

tween worker connections follow an EDF, we offer Corollary 1.

Corollary 1 *Let T be a positive time period. Then connections from P volunteer computing workers can be modeled as a Poisson process with rate parameter $\lambda = P/T$.*

Fundamentally, this means that we can handle worker communication unreliability by treating the group of all workers as a Poisson process with a tunable connection rate parameter. Based on the results shown in Fig. 3, we maintain that Corollary 1 is correct for short reconnection periods ($\leq$ 12 h) with long interval lengths ($\geq$ 5 min). These are well within the range expected in low latency batch computing. With greater numbers of workers, Corollary 1 may apply for longer T and shorter L. The idea of a tunable Poisson process is used in Section 4 to develop algorithms for maintaining a continuous stream of workers for computing low latency batches.

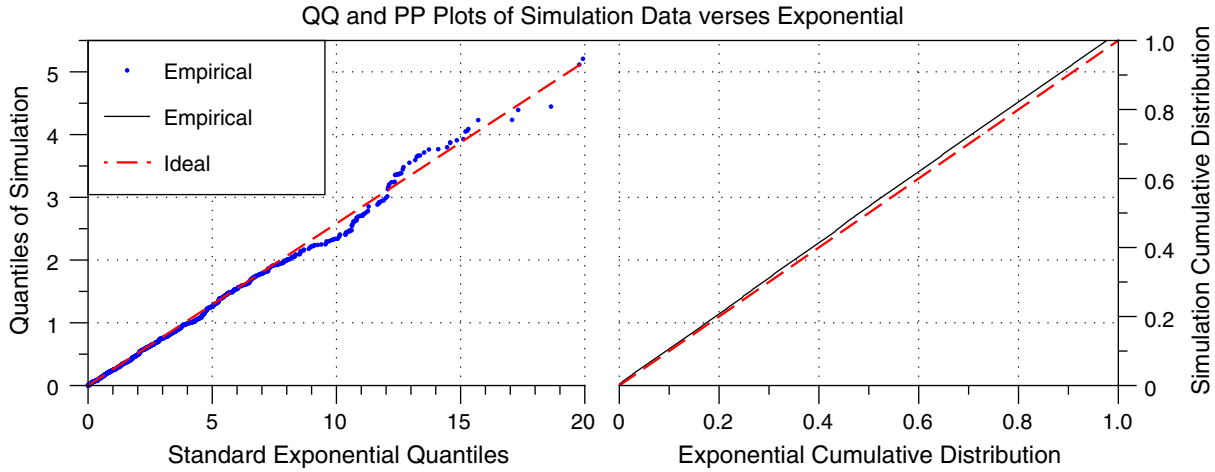

Fig. 4 Task and batch satisfaction rates from the experiments

3.4 Modeling VC Worker Computation

Here we propose a model of worker computation reliability based on worker availability prediction. Previous studies [13, 17] demonstrated techniques to ensure future worker availability based on correlated worker behavior, Bayesian classifiers and other techniques. These techniques provided means of organizing workers into groups based on training data for performing simultaneous computation. In this study, we use a simpler method of recent worker availability, which is more suited for low latency batches where workers need not be simultaneously active.

Although the other methods could possibly be used for low latency batch computing, we chose not to for a couple reasons. First, the other two methods discard roughly half of the worker population and require weeks of training data, which can discourage many VC projects from using them. Also, by using only highly available and predictable workers, the algorithms can cause cyclical unavailability in workers which share CPU time between projects in a round-robin fashion. Therefore, in this paper we focus on the technique described below.

To examine worker predictability, we performed 1 million simulations using the trace data subset described earlier. The goal of these simulations is to determine how well a workers recent past state can predict the future availability. Our hypothesis is that worker availability/unavailability can be predicted based on periodic worker behavior. Each simulation involves randomly selecting a worker with a chance proportional to its lifespan. The availability state of the worker is examined at a randomly selected time R in the workers lifespan. Next, for a range of time lengths T we examine worker availability at times $R - TX$ for $X \in [1, 10]$. The same analysis was performed for worker unavailability prediction based on previous unavailable states. The results of these analyses are shown in Fig. 5.

Figure 5a shows the accuracy of using the recent past availability to predict current availability and Fig. 5b shows the same for unavailability. For a given time period T and number of recent (un)available states X, these graphs show how accurate a prediction of (un)available is. For almost all of the simulations, (un)availability at all recent time periods is a strong indicator that the worker will be (un)available at time R.

In both graphs, shorter time periods yield better accuracy for predicting worker state. Figure 5a shows that using older states gives a poor prediction of worker availability. An interesting feature in this graph is the jump in accuracy for time periods of 1 day, implying that daily usage patterns exist for workers. Worker unavailability is also well predicted with time periods of 1 week and 3

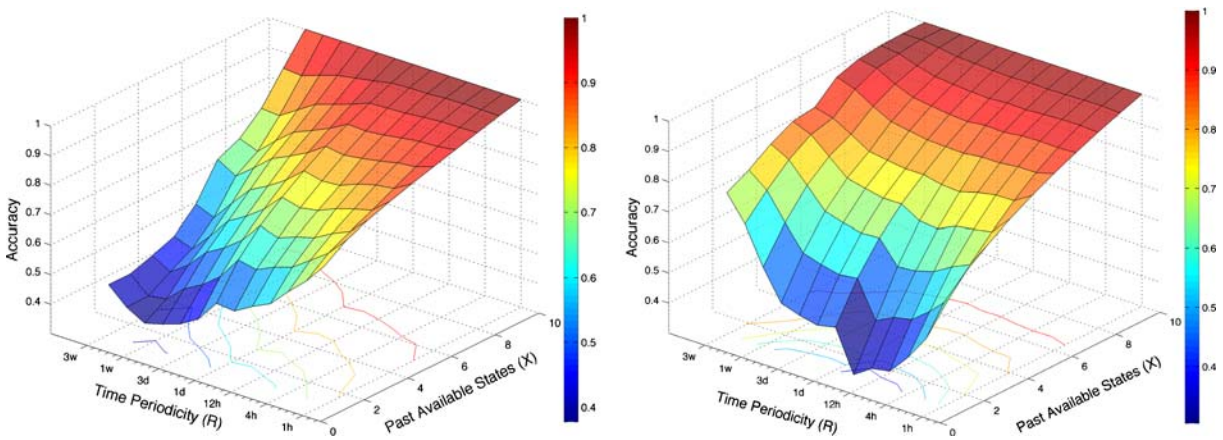

Fig. 5 Simulation results for worker availability and unavailability prediction. **a** Worker availability prediction results. **b** Worker unavailability prediction results

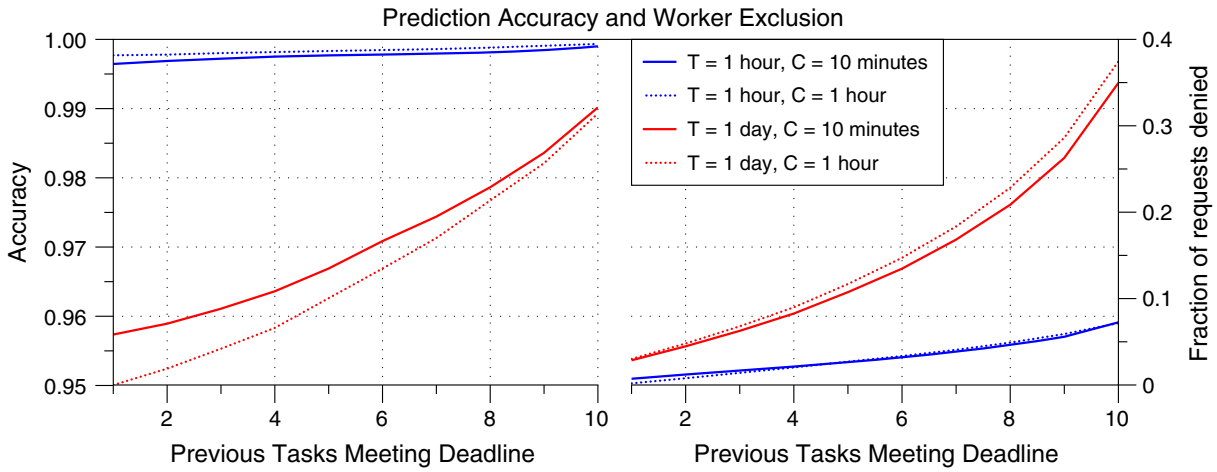

Fig. 6 Results of simulations for validating worker performance prediction

weeks, which would capture night and weekend downtime. We use these results in Section 4.3 to predict whether a worker will finish an assigned task before a deadline.

To confirm these results, we performed 100,000 simulations to examine the accuracy of using past worker availability to predict task completion success. In these experiments, we randomly select a worker and a time R when the worker is active. We simulate the worker receiving 10 tasks at times $R - TX$ for $X \in [1, 10]$ with each task having deadline $R - TX + 2C$ (twice the task computation time). We then check whether the worker completes the task by the deadline.

Figure 6 shows the results of the simulations. The left graph shows the accuracy of the past simulation results in predicting task completion in the present. In this graph we show the accuracy of predicting deadline satisfaction given the number of tasks which met their past deadlines. The lower axis shows the minimum number of past simulations that must have satisfied the deadline (out of 10) in order for a current prediction of "Will Meet Deadline". As seen, the accuracy of this method increases as the bounds on past failures get tighter.

The right graph shows the tradeoff of tightening the bounds for prediction. As the bounds get tighter, the fraction of failure predictions and thus the number of denied work requests grows. This is equivalent to a worker connecting to the master but not receiving a task due to poor past performance.

4 Task Distribution Algorithms

In order to meet batch deadlines, tasks must be distributed to workers in a timely manner. Pull-style VC task requests go from workers to the master. Thus, a sufficient number of task requests must occur between the batch submission and deadline. In a VC system, this is done by requesting workers to connect at certain times. This is known as the "reconnection time" and is denoted R_j for worker W_j.

A simple method for performing low latency batches is to estimate the submission time S_i of each batch B_i and request a number of workers to connect soon after that. However, even for synchronous batch computing, due to the unreliable nature of VC workers it is nearly impossible to predict when a given batch will complete and the next batch will begin. This is particularly true when a worker fails to complete a task and delays the generation of the next batch. For this reason, we assume that submission times cannot be known accurately far in advance.

In this section, we describe algorithms for ensuring a high probability of sufficient task requests to complete all batches before their deadlines.

We start by developing an algorithm for fully reliable workers, then modify it to handle communication and computation unreliability. Section 4.1 presents an algorithm for fully reliable (communication-reliable and computation-reliable) workers, and proves that it satisfies all deadlines in certain conditions. In Section 4.2 we provide an algorithm for semi-communication-reliable workers with a probabilistic bound on failure. The algorithm for semi-reliable (semi-communication-reliable and semi-computation-reliable) workers is given in Section 4.3, and also provides a probabilistic bound on failure. The effectiveness of these algorithms is demonstrated in Section 5.

4.1 Fully Reliable Homogeneous Workers

In this section we consider fully reliable workers, with guaranteed communication and computation times. Using fully reliable workers, we develop an algorithm for task distribution that provides a basis for later algorithms with unreliable workers. In this section, workers are considered computationally homogeneous and reliable, meaning that a task always takes C seconds and finishes at time $R_j + C$. To meet the deadline D_i, all tasks in B_i must be distributed to workers before the distribution deadline $L_i = D_i - C$. We assume that all workers connect before S_1.

Algorithm 1 shows the algorithm for the master when dealing with fully reliable workers. The algorithm works by pre-assigning a task to a worker using the variables *AssignBatch* and *AssignTask*. Worker W_j connects at time R_j or immediately if the current time is past R_j. Upon connection the worker receives a task (line 5) and next connection time, then executes the task. The master gives workers reconnection times that ensure the workers will receive tasks at the necessary time. Because all workers are homogeneous and reliable, task assignment is in a simple round-robin fashion and the algorithm needs only ensure that N task requests occur during each batch. It is worth noting that the calculation of R_j (line 12) does not simply assign the batch submission time S_i to a worker. Instead, the algorithm spreads worker connections over the entire batch. This prevents the master from being overwhelmed by connections, and is also necessary in later algorithms for semi-reliable workers. Given this algorithm and constraints on the task length, deadline/submission times, and number of tasks/workers, Theorem 1 proves that all batches will meet their deadlines.

Algorithm 1 Fully Reliable Homogeneous Workers

```
1:  AssignBatch ← 1, AssignTask ← 1, SendBatch ← 1, SendTask ← 1
2:  while SendBatch < M do
3:      Get connection from W_j
4:      if CurrentTime() ≥ S_SendBatch then
5:          Send task T^SendBatch_SendTask to W_j
6:          SendTask ← SendTask + 1
7:      end if
8:      if SendTask > N then
9:          SendTask ← 1, SendBatch ← SendBatch + 1
10:     end if
11:     if AssignBatch ≤ M then
12:         R_j ← S_AssignBatch + (AssignTask − 1) ∗
            (L_AssignBatch − S_AssignBatch)/(N − 1)
13:         AssignTask ← AssignTask + 1
14:     end if
15:     if AssignTask > N then
16:         AssignTask ← 1, AssignBatch ←
            AssignBatch + 1
17:     end if
18: end while
```

Theorem 1 *If, for all batches, the deadline time is greater than the submission time plus the maximum execution time among workers* ($\forall i$, $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$) *then Algorithm* 1 *results in all batches meeting their execution deadlines.*

Proof Proof by induction. At time S_0, by definition no workers are executing a task and all P workers have connection times. Next, assume that no workers are executing a task and all workers have been given connection times by time S_i. We now demonstrate that if $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$, then all tasks in B_i will be finished at or before D_i and at time S_{i+1} there will be no workers executing tasks.

If $P \geq N$ then at time S_i all tasks for batch B_i have been assigned, and each worker will receive at most 1 task from the batch. The latest task distribution time will be $S_i + (N-1)\frac{L_i - S_i}{N-1} = L_i = D_i - C$ and the latest task completion time will be D_i, meaning that no workers are executing a

task after D_i. If $D_i < S_i + C$, then $L_i < S_i$, which is a contradiction because no tasks from B_i can be distributed before S_i.

If $P < N$ then at time S_i only P tasks from batch B_i have been assigned. During the execution of batch B_i, a worker will request and execute either $\lfloor \frac{N}{P} \rfloor$ or $\lceil \frac{N}{P} \rceil$ tasks. Because a worker executes tasks one by one, the latest a worker will request a task is at time $max(L_i, C(\lceil \frac{N}{P} \rceil - 1))$ and the latest task completion time will be $max(D_i, C\lceil \frac{N}{P} \rceil)$. If $D_i < S_i + C\lceil \frac{N}{P} \rceil$, the batch completion time will be $C\lceil \frac{N}{P} \rceil > D_i$ and the deadline will not be met. In this case, there can be no guarantees about the execution state of workers at time S_{i+1}. If $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$, the batch completion time will be D_i and no workers will be executing tasks at S_{i+1}.

Therefore all batches will meet their execution deadlines if and only if $\forall i$, $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$. □

4.2 Semi-Communication-Reliable Homogeneous Workers

Algorithm 2 Semi-Communication-Reliable Homogeneous Workers

1: Calculate λ from K and N; estimate P; $T \leftarrow \frac{0.9PL}{\lambda}$
2: $SendBatch \leftarrow 1$, $SendTask \leftarrow 1$
3: **while** $SendBatch < M$ **do**
4: Get connection from W_j
5: **if** $CurrentTime() \geq S_{SendBatch}$ **then**
6: Send task $T^{SendBatch}_{SendTask}$ to W_j
7: $SendTask \leftarrow SendTask + 1$
8: **end if**
9: $R_j \leftarrow CurrentTime() + T$; Send R_j to W_j
10: **if** $SendTask > N$ **then**
11: $SendTask \leftarrow 0$, $SendBatch \leftarrow SendBatch + 1$
12: P_{active} = num connections in last $2T$ seconds, $T \leftarrow \frac{0.9P_{active}L}{\lambda}$
13: **end if**
14: **end while**

In this section we consider homogeneous workers that are computation-reliable and semi-communication-reliable. In other words, they behave like VC workers when requesting tasks, but will always complete a task on time once it is received. Here we modify Algorithm 1 to use the model of worker requests from Section 3.3 and ensure enough task requests for a given batch.

Predicting the availability state of an arbitrary VC worker at a specific time is extremely difficult, especially for times far in the future. Algorithm 1 cannot be used in such environments because it requires each worker W_j to be available at R_j. As previously mentioned, Algorithm 1 does not request all workers to connect at time S_i but instead maintains a constant rate of task requests between S_i and L_i. In the same way, computing low latency batches with semi-communication-reliable workers is possible by maintaining a stream of task requests.

In Section 3.3, we demonstrated that task requests from VC workers can be modeled as a Poisson process. Given this model, we now determine how to calculate the reconnection period T so as to distribute all tasks before the batch deadline L. Given Corollary 1, we can control the probability K of at least N task requests being sent to the master from P workers in a time period L. This probability is controlled by specifying a reconnection period T based on P, L and N. Although in this algorithm we use P to calculate T, there is no intrinsic dependence of T on P. Other techniques for counting active workers are just as suitable, as long as the active time period is greater than T.

The number of task requests occurring in a Poisson process follows the Poisson distribution. This gives the probability of exactly N task requests occurring in a given time period. Because this is a probabilistic model we can only put a bound on the probability K of a specified number of task requests occurring. For a probability K of at least N task requests in a given time period L, λ must satisfy:

$$K \geq 1 - \sum_{i=0}^{N-1} \frac{e^{-\lambda}\lambda^i}{i!} \tag{1}$$

Based on our observations, roughly 10% of workers become inactive in a daily cyclical pattern. λ is calculated assuming large numbers of workers do not become inactive simultaneously. To compensate, we calculate the reconnection period using 90% of the active worker count. In Section 3.3 we showed worker connections can be modeled as a Poisson process with rate parameter $\lambda = P/T$.

Given λ from the above equation we can rewrite this as the reconnection period:

$$T = \frac{0.9PL}{\lambda} \tag{2}$$

The number of active workers P can change over long time periods, though λ and L are assumed to be constant. Therefore the value of T must change during the course of the computation. One way to track active workers is to count the number of workers which connected recently, in this case, the number of unique workers which connected in the last $2T$ seconds.

Algorithm 2 demonstrates how to use the active worker count to distribute tasks to semi-communication-reliable workers. This algorithm ensures sufficient task requests to the master before the distribution deadline even with individual worker unreliability and daily fluctuations. Algorithm 2 differs from Algorithm 1 in that it does not track the pre-assignment of tasks to workers. Instead, the goal of N task requests is implicitly achieved by altering the reconnection rate (line 12). We demonstrate the effectiveness of this algorithm in Section 5.

4.3 Semi-reliable Heterogeneous Workers

Finally, we propose an algorithm to replicate and distribute tasks to semi-reliable heterogeneous workers. These are semi-communication-reliable and semi-computation-reliable workers as described in Section 2 and modeled in Sections 3.3 and 3.4. As demonstrated, worker availability at time R can be estimated based on the number of times the worker was available at past times $R - TX$.

To determine whether a worker will complete a given task by the deadline, we estimate the probability of the worker providing the required amount of computational power between task distribution and the deadline. Suppose a worker receives a task at time R with deadline D. For a worker with task computation time C, we want to estimate the probability of the worker being in the available state for more than C seconds in the time interval $[R, D]$. If $C > D - R$ then the probability is 0. Otherwise, we estimate the probability based on the time to compute the task if it were started at past times $R - TX$ using $T = 1$ day and $T = 1$ week. If more than half of the tasks computed at the previous time periods would have missed the deadline, $Pr_i^{Success} = 0$ and the worker is not assigned a task. Otherwise, the probability is estimated based on mean worker task computation time $\overline{C}$ relative to this workers computation time C_i using the equation $Pr_i^{Success} = min(0.99, 2\overline{C}/C_i)$. The justification for this is that for faster workers, small unpredictable periods of unavailability will have less effect and the task will more likely finish before the deadline. This type of speed based scheduling is also investigated in [15].

Algorithm 3 shows the master task distribution algorithm for semi-reliable heterogeneous workers. This algorithm is similar to Algorithm 2, except we create replicas of some tasks that have a low probability of finishing before the deadline. To decide which tasks to replicate, we keep an estimate of the probability Pr_i^{Fail} of missing the deadline for each task T_i. This estimate starts at 1 for all tasks, then is updated (line 9) based on the estimated probability of success (line 6) as the tasks are assigned to workers. Also, because the workers are heterogeneous, L is computed using the mean task completion time $\overline{C}$.

Algorithm 3 Semi-Reliable Heterogeneous Workers

1: Calculate λ from K, N; estimate P; $T \leftarrow \frac{0.9PL}{\lambda}$
2: $SendBatch \leftarrow 1$, $SendTask \leftarrow 1$
3: $\forall i \in [1, SendTask]$, $Pr_i^{Fail} = 1$
4: **while** $SendBatch < M$ **do**
5: Get connection from W_j
6: Calculate probability $Pr^{Success}$ of worker finishing task before deadline
7: **if** $CurrentTime() \geq S_{SendBatch}$ and $Pr^{Success} > 0$ **then**
8: Send task T_i with highest Pr^{Fail} to W_j
9: $Pr_i^{Fail} \leftarrow Pr_i^{Fail}(1 - Pr^{Success})$
10: **end if**
11: $R_j \leftarrow CurrentTime() + T$; Send R_j to W_j
12: **if** All tasks finished **then**
13: $SendBatch \leftarrow SendBatch + 1$
14: $\forall i \in SendTask$, $Pr_i^{Fail} = 1$
15: $P_{active} =$ num workers in last $2T$ seconds; $T \leftarrow \frac{0.9P_{active}L}{\lambda}$
16: **end if**
17: **end while**

Each of the algorithms described here distributes N tasks from each of M batches. At the end of each batch, the number of active workers is calculated, which takes $O(P)$ (or less, depending on the method). For Algorithms 1 and 2, distribution of a task takes constant time $O(1)$, so these algorithms have time complexity $O(M(N + P))$. For Algorithm 3, the task with the highest probability of failure is sent to the worker. Using a tree structure, maintaining a list of tasks sorted by probability of failure has time complexity $O(logN)$. Therefore Algorithm 3 has time complexity $O(M(NlogN + P))$.

However, it is worth pointing out that for the numbers of tasks and workers typically involved in VC systems ($N < 10^5$ per day, $P < 10^7$ [1]), each task distribution requires very little CPU time. In fact, the limiting factor is generally bandwidth to the master rather than CPU time.

5 Experiments

We conducted a series of simulation experiments to test the algorithms in Sections 4.2 and 4.3 and compare them to alternate algorithms. Simulations were implemented using a custom event-driven simulator program that emulates a VC master-worker environment based on trace files using double precision for all times. To improve simulation time, workers are skipped forward to a week before the simulation start. Master operation depends on the algorithm being executed. Workers execute tasks until completion—they do not abort a task if the deadline has past. The trace files for all simulations in this section consisted of 37,472 randomly selected workers from the trace data set in Section 3.1.

5.1 Semi-Communication-Reliable Workers

To confirm that Algorithm 2 provides sufficient task requests from workers, we performed experiments using the trace data described above. The parameters for the experiments are shown in Table 2. These parameters represent a range of possible low latency applications, from small to large batches with short to long tasks. Simulations with impossible parameter combinations (e.g. $C \geq D$) were not performed. Dates for simulation start (S_0) were chosen to get a good range of active workers.

Table 2 Simulation parameters

Parameter name	Parameter value
Number of batches (M)	256
Tasks per batch (N)	1024, 2048, 4096
Target success probability (K)	0.99
λ derived from K, N	1100, 2155, 4247
Batch deadline (D)	1 h, 4 h
Simulation start (S_0)	Sep 1, 2007; Nov 1, 2007; Jan 1, 2008
Batch submission (S_i)	$S_{i+1} = S_i + D$
Task computation time (C or $\overline{C}$)	30 min, 1 h, 2 h

Figure 7 shows two sample results from the experiments. The figures show the active worker count at the start of each batch, and the number of task requests received during each batch. The light histogram represents the number of task requests that arrived before the batch computation deadline, while the dark histogram represents the number of task requests that arrived before the batch distribution deadline. The horizontal line is the minimum number of task requests needed to successfully complete the batch. Daily and weekly worker unavailability cycles can be clearly seen in the active worker count for both graphs.

These results show that Algorithm 2 maintains a steady stream of worker task requests for batches with varying characteristics. Even though the number of active workers changes significantly over time, the required number of task requests arrive before or occasionally slightly after the distribution deadline. It is worth noting that task requests are spread evenly before and after the distribution deadline. This is important for systems where the submission time of a future batch is unknown, and a constant stream of task requests is required to ensure batch satisfaction.

For comparison, we also implemented a simpler algorithm that adjusts the reconnection rate using a feedback loop based on the fraction of successful task requests in the previous batch. We refer to this algorithm as the "shifting" algorithm. If a batch fails to receive enough task requests before the distribution deadline, this algorithm decreases the reconnection rate proportional to the number

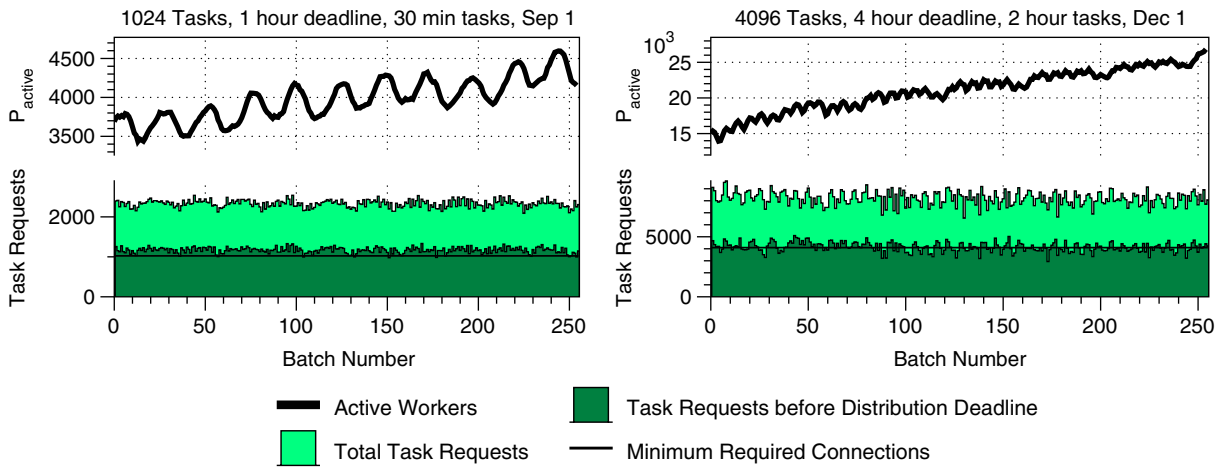

Fig. 7 Two sample results from semi-communication reliable worker experiments

missed. If a batch receives over 10% more than the needed task requests, it increases the reconnection rate by a proportionate amount.

Figure 8 shows a comparison of the shifting reconnection rate algorithm with the the Poisson based algorithm described in Section 4.2. This shows the percent of batches in the experiments which received a given fraction of task requests before the distribution deadline. In over 70% of batches, both algorithms distributed all of the

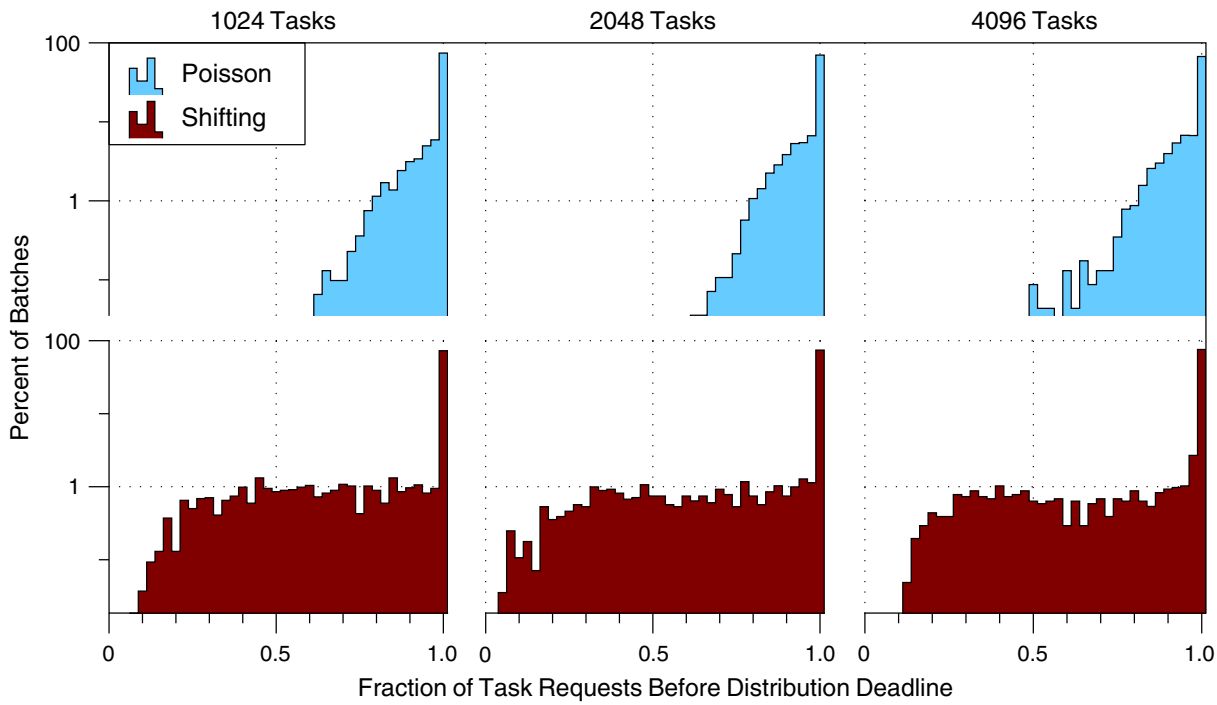

Fig. 8 Fraction of tasks in a batch that were distributed before the distribution deadline, comparing two algorithms. From *left* to *right*, the graphs represent batches with 1024, 2048 and 4096 tasks

tasks before the deadline. However, the shifting algorithm resulted in many more batches with very few task requests. This is because the shifting algorithm can only react to changes in the active worker count after they have affected a batch, as opposed to the Poisson method which adapts to the worker count as well as the expected fluctuation. We found that with long reconnection periods, the shifting algorithm increases the reconnection rate for multiple successive batches, then suddenly encounters a lack of workers as the previous adjustments take effect. This helps explain the batches with very few task requests for the shifting algorithm in Fig. 8.

From these results, we feel confident that Algorithm 2 and the model described in Section 3.3 are useful and valid for ensuring sufficient task requests to meet low-latency batch deadlines in VC systems. Although the algorithm failed to provide complete reliability in terms of task requests, we demonstrated that it can adapt to changes in worker availability better than a simpler algorithm.

5.2 Semi-Reliable Workers

Finally we test the efficacy of Algorithm 3 in completing batch tasks before the deadline. The parameters for these experiments are the same as Section 5.1, shown in Table 2. In all these experiments, an average of two copies of each task is distributed to workers (more or less depending on the algorithm). For comparison, we also tested two alternate methods of computing the probability of success $Pr^{Success}$ in Algorithm 3.

In these experiments we tested three ways of computing $Pr^{Success}$. The first involves simple replication where each task is replicated twice and sent to an arbitrary worker, regardless of worker speed or past history. The second involves calculating $Pr^{Success}$ solely based on worker speed, using the equation described in Section 4.3. The final technique, described in Section 4.3 uses predictions based on past worker history in combination with worker speed to estimate $Pr^{Success}$.

The results of the experiments for Algorithm 3 are shown in Fig. 9. This figure shows the percent of batches where a given fraction of task requests satisfied the deadline. The graphs going left to right represent different ratios of task computation time to deadline time. The leftmost graphs represent all experiments with the tighter deadline of task computation time C being half of deadline time D. The rightmost graphs represent experiments with a looser deadline of $C = D/8$. From top to bottom, the graphs show the results of performing Algorithm 3 using the simple method of task replication and distribution, task assignment based on worker speed, and task assignment based on worker speed with past history.

First we notice that longer deadlines relative to task size result in more tasks and batches being satisfied. This makes sense because Pr^{Fail} is set to 0 when a task completes, meaning the remaining task requests in the batch will more likely be assigned tasks that would fail anyway. In contrast, with tighter deadlines the tasks are allocated to workers in a less efficient manner in order to meet the deadline.

There is a significant jump in batch satisfaction from using worker speed to estimate $Pr^{Success}$. This effect has been noted before in desktop grids [15]. The same study found no connection between worker speed and availability. This means that for any availability pattern a faster worker will have a greater chance of meeting a deadline simply because the task is more likely to finish before the worker goes into a long unavailable interval.

The bottom three graphs in Fig. 9 show the effectiveness of the techniques proposed in this paper. For looser deadlines of $C = D/4$ and $C = D/8$, our techniques result in over 90% of batches being satisfied. For the tighter deadline of $C = D/2$, the number of satisfied batches significantly drops but is still higher than the alternate techniques. With tighter deadlines, worker unavailability from unpredictable causes such as user activity becomes more of an issue.

Based on these results, we feel Algorithm 3 provides a good way of performing low latency batches in a VC environment. Compared to simpler methods of managing communication unreliability such as the shifting reconnection algorithm, the Poisson method described in this paper provides a more accurate method of controlling workers. For managing computational unreliability,

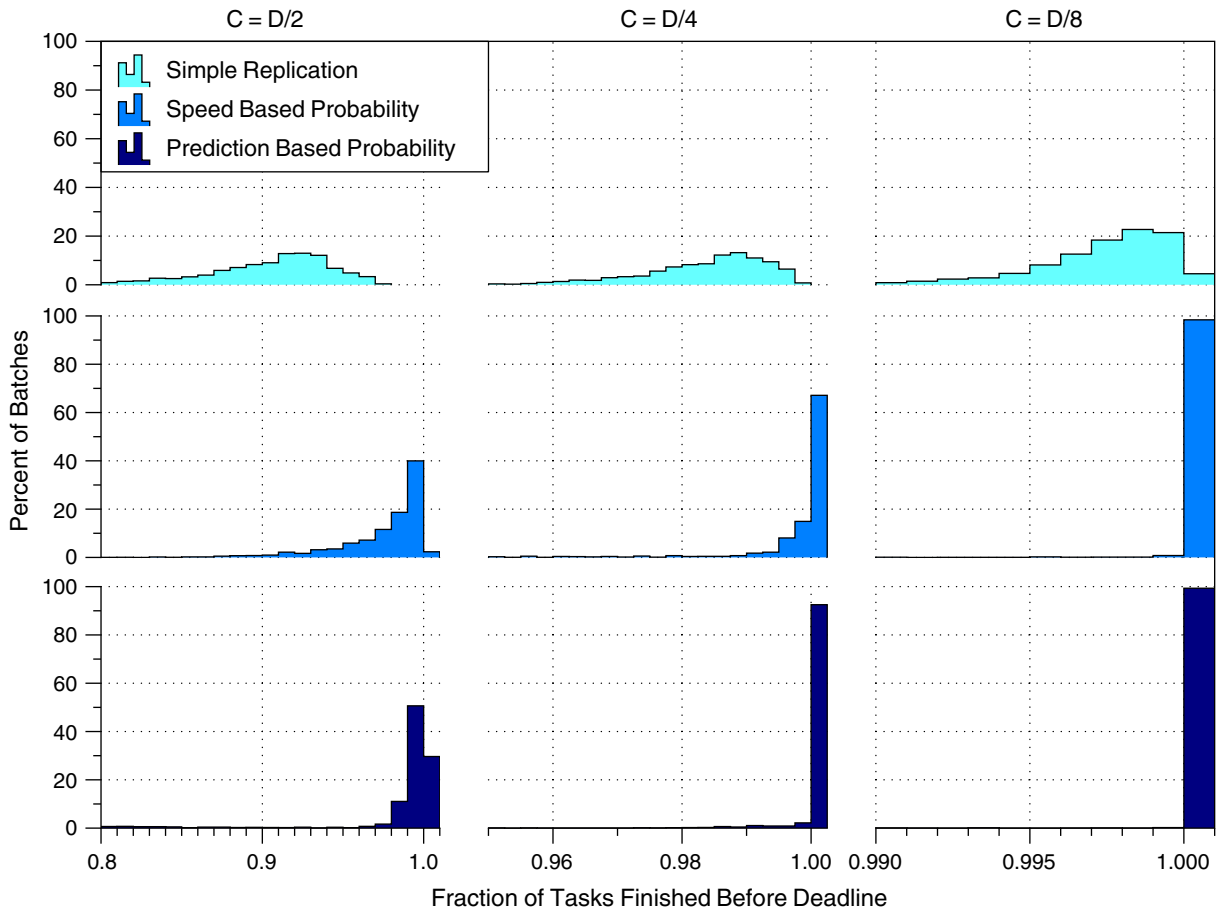

Fig. 9 Task and batch satisfaction rates from the experiments

probabilistic assignment based on worker speed and past history proves more effective than arbitrary task assignment or using worker speed only.

6 Related Work

Related studies examined task distribution in grid and VC environments [5–7], though some of these assume a task push model and are not valid for pull-style VC environments. Others describe methods to maximize total system throughput rather than meet specific task deadlines. There are also several works [8, 9, 13, 15–17, 21] analyzing the characteristics of VC and desktop grid environments which are applicable to high throughput computing but don't target low latency style VC. To the best of our knowledge, this paper is the first to investigate methods for computing low latency batches in a pull-style VC environment.

There is work similar to our study in regards to completing batches of tasks with deadlines [22], though this focused on desktop grid environments rather than volunteer computing. This study viewed the system as a buffer with batches of tasks periodically entering and expiring from the buffer. The authors analyzed the appropriate buffer size to ensure maximum task completion rates in a desktop grid environment, where tasks can be assigned to arbitrary workers. Future work in low latency batch computing could use this type of buffer, especially for computations performing multiple simultaneous batches.

One technique for handling unreliable workers is the use of checkpointing or "heartbeats" to the master server [23, 24]. In this technique, workers

periodically report and/or save their progress to the master server. This allows for duplication of tasks which are unlikely to meet the deadline. However, we believe that this would not be useful in low latency because of the short task computation time. For short running tasks, intelligent scheduling will provide better results than checkpointing.

Other research related to VC scheduling includes using evolutionary algorithms to develop scheduling algorithms [25] and using P2P to perform load balancing in VC systems [26]. A possible future field of research is to use genetic algorithms in predicting worker availability for low latency batch computing, though other methods [13, 17] appear to already be very successful and more applicable to low latency computing.

7 Conclusion and Future Work

In this paper we proposed methods for performing low latency batches of tasks with deadlines in VC environments. To do so, we first proposed analysis based models for handling communication and computation unreliability in VC workers. These models were used to develop task distribution algorithms aimed at low latency batch VC, which were then validated using execution trace data from an actual VC environment.

Although the experiments in Section 5 showed the effectiveness of these algorithms, further work can be done to improve the algorithms, especially in regards to estimating task success on a given worker. Other techniques [13, 17] show promise in this regard, though implementations should avoid computationally intensive techniques when computing with deadlines.

To make practical use of these results we plan to study methods for integrating low-latency batch computing in existing VC systems such as BOINC. BOINC already supports client reconnection times and recording past availability states, so implementing low latency in BOINC is primarily a matter of allowing users to specify batch characteristics and quickly processes completed batches.

Acknowledgements Thanks for Prof. Noriyuki Fujimoto for assistance in this research. This work was supported in part by Research Fellowship (19·55401), and Grant-in-Aid for Scientific Research (A)20240002 from the Japan Society for the Promotion of Science, and by the Global COE Program “in silico medicine” at Osaka University.

References

1. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002)
2. Larson, S.M., Snow, C.D., Shirts, M., Pande, V.S.: Folding@home and genome@home: using distributed computing to tackle previously intractable problems in computational biology. In: Modern Methods in Computational Biology. Horizon, Marseille (2003)
3. Valiant, L.: A bridging model for parallel computation. Commun. ACM **33**(8) (1990)
4. Schopf, J.M., Berman, F.: Stochastic scheduling. In: Supercomputing ’99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM), p. 48. ACM, New York (1999)
5. Budati, K., Sonnek, J., Chandra, A., Weissman, J.: Ridge: combining reliability and performance in open grid platforms. In: HPDC ’07: Proceedings of the 16th International Symposium on High Performance Distributed Computing, pp. 55–64. ACM, New York (2007)
6. Kondo, D., Chien, A.A., Casanova, H.: Resource management for rapid application turnaround on enterprise desktop grids. In: SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 17. IEEE Computer Society, Washington, DC (2004)
7. Rood, B., Lewis, M.J.: Scheduling on the grid via multi-state resource availability prediction. In: 9th IEEE/ACM International Conference on Grid Computing, 2008, pp. 126–135 (2008)
8. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: an empirical study. In: 2004 Proceedings of the 18th International Parallel and Distributed Processing Symposium, p. 26 (2004)
9. Malecot, P., Kondo, D., Fedak, G.: Xtremlab: a system for characterizing internet desktop grids. In: 2006 15th IEEE International Symposium on High Performance Distributed Computing, pp. 357–358 (2006)
10. Golle, P., Mironov, I.: Uncheatable distributed computations. In: Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer’s Track at RSA, vol. 2020, pp. 425–440 (2001)

11. Sonnek, J., Chandra, A., Weissman, J.B.: Adaptive reputation-based scheduling on unreliable distributed infrastructures. IEEE Trans. Parallel Distrib. Syst. **18**(11), 1551–1564 (2007)
12. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 4–10. IEEE Computer Society, Washington, DC (2004)
13. Kondo, D., Andrzejak, A., Anderson, D.P.: On correlated availability in internet-distributed systems. In: 9th IEEE/ACM International Conference on Grid Computing, 2008, pp. 276–283 (2008)
14. Anderson, D.P., Fedak, G.: The computational and storage potential of volunteer computing. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 73–80. IEEE Computer Society, Washington, DC (2006)
15. Kondo, D., Fedak, G., Cappello, F., Chien, A.A., Casanova, H.: Resource availability in enterprise desktop grids. Future Gener. Comput. Syst. **23**(7), 888–903 (2007)
16. Nurmi, D., Brevik, J., Wolski, R.: Modeling machine availability in enterprise and wide-area distributed computing environments. In: Euro-Par05, pp. 432–441 (2005)
17. Andrzejak, A., Kondo, D., Anderson, D.P.: Ensuring collective availability in volatile resource pools via forecasting. In: DSOM '08: Proceedings of the 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pp. 149–161. Springer, Berlin (2008)
18. Heien, E., Fujimoto, N., Hagihara, K.: Computing low latency batches with unreliable workers in volunteer computing environments. In: IEEE International Symposium on Parallel and Distributed Processing. IPDPS 2008, pp. 1–8 (2008)
19. Stephens, M.A.: Edf statistics for goodness of fit and some comparisons. J. Am. Stat. Assoc. **69**(347), 730–737 (1974)
20. Parzen, E.: Stochastic Processes. Society for Industrial and Applied Mathematics, Philadelphia (1999)
21. Byun, E., Choi, S., Baik, M., Hwang, C., Park, C., Jung, S.Y.: Scheduling scheme based on dedication rate in volunteer computing environment. In: The 4th International Symposium on Parallel and Distributed Computing. ISPDC 2005, pp. 234–241 (2005)
22. Kondo, D., Kindarji, B., Fedak, G., Cappello, F.: Towards soft real-time applications on enterprise desktop grids. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 65–72. IEEE Computer Society, Washington, DC (2006)
23. Kondo, D., Araujo, F., Domingues, P., Silva, L.: Validating desktop grid results by comparing intermediate checkpoints. Technical Report TR-0059 (2006)
24. Christensen, C., Aina, T., Stainforth, D.: The challenge of volunteer computing with lengthy climate model simulations. In: First International Conference on e-Science and Grid Computing, pp. 8–15 (2005)
25. Estrada, T., Fuentes, O., Taufer, M.: A distributed evolutionary method to design scheduling policies for volunteer computing. In: CF '08: Proceedings of the 2008 Conference on Computing Frontiers, pp. 313–322. ACM, New York (2008)
26. Murata, Y., Inaba, T., Takizawa, H., Kobayashi, H.: Implementation and evaluation of a distributed and cooperative load-balancing mechanism for dependable volunteer computing. In: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC. DSN 2008, pp. 316–325 (2008)